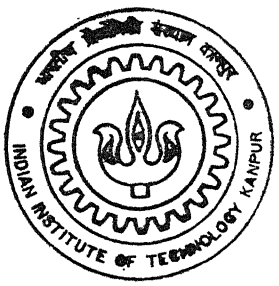
A Technique for Recovering Grammar from Legacy Programs

By

Shiladitya Biswas



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Indian Institute of Technology Kanpur

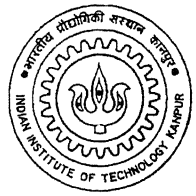
APRIL, 2003

A Technique for Recovering Grammar from Legacy Programs

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

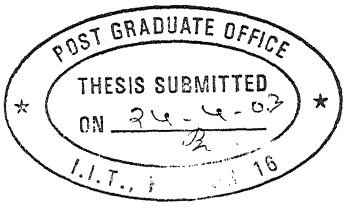
Shiladitya Biswas



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

April 2003



Certificate

This is to certify that the work contained in the thesis entitled "*A Technique for Recovering Grammar from Legacy Programs*", by *Shiladitya Biswas*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

April 2003

(Dr. Sanjeev Kumar Aggarwal)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Automation is the key to productivity gains in software reengineering. Many tools like Test Coverage Analyzer, Static Analyzer and Code Compliance Checker are not only required for quality management but also for adding new features to existing systems. Most of these require grammar of the language in which a program is written. It is quite obvious that any program analysis tool would essentially be grammar based. Grammar is usually available for a standard language but not for its various dialects that are used to write proprietary software. In this thesis we propose an approach for extracting the grammar from a set of programs that is a variant of a known language. We do not attempt to construct grammar from scratch. We try to recover the complete grammar by starting with an approximate grammar and set of programs. We propose an iterative method by which we parse the source program to locate the missing constructs and then incorporate the missing grammar rules to obtain the complete grammar.

Acknowledgements

I take this opportunity to express my sincere thanks and immense gratitude to my supervisor Dr. Sanjeev Kr. Aggarwal for his invaluable guidance and constant support during the course of this thesis. His experience and farsightedness prevented me from getting diverted from the problem at hand and allowed me to complete my thesis successfully.

I would also like to thank the entire faculty and staff of CSE Department at IIT Kanpur. I would also like to thank all my batch mates who have made my stay here a really memorable experience. I am specially grateful to my friends Diganchal, Ranjit, Gopi Kishore, Sandip, Debi Prosad, Vijay and Imtiaz for making my stay at IIT Kanpur a memorable one. Vaibhav, Kapil, Subhash and Sreenu provided a great atmosphere in CS301.

My family has been a source of constant support and inspiration for me. My parents and my brother Somtirth deserve the greatest credit for this work. This thesis has been possible because of their love and blessings.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A major time and effort of software engineers is spent in the enhancement and renovation of existing software systems. Software testing, debugging, maintenance, understanding legacy systems, reverse engineering and reengineering all require automated tool support. All these activities involve working with large amounts of code where often the person who had originally written the code is not present. This problem is more prevalent for legacy systems. There are other problems also in working with legacy systems, such as use of outdated development methods, extensive patches and modifications to the software and outdated or missing documentation. So, further evolution and development may be prohibitively expensive. We need to study legacy systems not only because they are valuable for us and need to be maintained but also to understand what problems are present in its design so that they can be avoided in future designs. This is called *Reverse Engineering* i.e. examine ways to recover design and analysis models from existing systems. Another related aspect is *Reengineering* i.e. exploring techniques to transform systems to make them more maintainable. Organizations reengineer existing software systems using newer languages, platforms and tools and then work with the reengineered system.

The need for updating and renovating business-critical software may also arise due to various reasons like change in business requirements, modernization of technological infrastructure, etc to name a few. These factors warrant the updating of the existing software system. In developing a new product or product line or migrating to modern software architecture, it is unrealistic to start from scratch. A realistic approach can start by analyzing legacy systems to understand the current architecture and

developing a strategy for reusing existing assets.

According to conservative estimates there are at least 500 languages and dialects available commercially or in public domain. Apart from that over 200 proprietary languages have been developed by corporations for their own use [8]. The programs written in these proprietary languages compose 30 percent of the total software. The worst part is for some of these languages there is no tutorial or manual available and there are only few trained professionals available. Languages play a crucial part in reengineering. There is a need to obtain the grammar of the language in which a software is written in order to construct tools for software reengineering.

1.1 Relevance of Grammar in Software Testing and Reengineering

The most crucial aspect of any programming language is its grammar. A grammar is a formal specification of the syntactic structure of a language. Grammar is essential in developing program analysis tools. Enhancing existing systems and reengineering require dealing with legacy systems. These legacy systems are generally very large in size. So, it is not feasible to analyze them manually. Some kind of tool support is essential to aid in the renovation and updating tasks that we discussed above. Automation is the key to productivity gains in software reengineering. The need for rapid development of tools that help in analysis and transformations of software systems is well established. The bottleneck for rapid development of such tools is constructing the parser and its connection to existing back-ends. In [5] Newcomb and Scott claim that language independent support cannot assess complex situations, and that even the best tools break down on multi-language systems, embedded sublanguages, and embedded data-manipulation languages.

Lot of research work has been done for automating the reverse engineering process [3, 14, 18, 21]. These have led to development of tools for recognizing the program's structure, procedures and components of the program, tools for analyzing and transforming programs. Tools like pretty printers, debuggers, animators, profilers, etc. are some examples. One common requirement that all these approaches have, is the grammar of the language in which the programs are written.

Tools for quality management like test coverage analyzer, code compliance checker, static analyzer are also useful in adding new features to existing systems. Most of these approaches and tools need the grammar of the language in which the programs are written. It is quite clear that the grammar of the language in which programs are written is essential for developing many useful tools, which assist in the analysis and maintenance of existing systems as well as in the development of new systems with high quality measures.

Grammars are omnipresent in software engineering. In other areas such as database design, flat files are designed using grammars [11]. Also document type definitions (DTDs) or XML schemata can be regarded as grammars. DTDs/XML are widely used to define the abstract representation of structured documents. In order to learn a new language also we need to refer to its grammar.

1.2 Motivation

From the previous discussion it is clear that grammar is an invaluable resource for developing automated tools for checking, maintaining and enhancing quality of large software systems. There is so much code around (around 7 billion function points [22]) that it is becoming prohibitively expensive in terms of both time and effort to deal with the existing codebase by hand. Although grammar for *"standard"* languages are known and widely available, it often happens that organizations use dialects of standard languages which have extra constructs. Very little documentation and technical support is available for these proprietary and obscure languages. So one cannot hope of getting grammar from the compiler front-end. Nevertheless, we wish to develop the aforementioned tools for these applications. Such tools can be developed only when the grammar for the language is available. There are many situations in which programs written in some language are available but the grammar for the language is not. So, there is a need to devise some approach for extracting grammar from programs.

1.3 Related Work

We can broadly divide grammar recovery approaches into three categories. The first is the engineering approach where grammar is extracted from various sources in different forms like language reference manuals, the code of CASE tools for the language, etc. Language reengineering deals with recovering grammar from these sources and using it for building tools. The second set of approach is more compiler based like using dynamic parsers and evolving grammars. Finally there are machine learning, AI techniques and natural language processing methods.

1.3.1 Language Engineering Approaches

Development of Language Descriptions

In [19] Alex Sellink and Chris Verhoef propose some tools that aid in the development, assessment and reengineering of language descriptions [1]. From an incorrect and incomplete grammar like the one given in a language reference manual, these tools can generate a correct and complete grammar that can be used for reengineering purposes.

The grammar for a language is itself written using some dialect of Backus-Naur Formalism(BNF) called *S*yntax Definition Language. So a document containing a language can be viewed as a program written in a syntax definition language. Such a document can be parsed by a syntax definition language parser like any other program written in a programming language. Creation of a parser amounts to defining the grammar of the metasyntax in BNF.

This attempt failed to come up with a working parser. The reason was because the on-line documentation contained much less than the actual language comprised. The approach also assumes availbility of complete grammar. Unfortunately this assumption does not hold true for most legacy applications.

[1] Grammar and Language Description are synonymous.

Semi Automatic Grammar Recovery

In [9] R. Lammel and Chris Verhoef propose an approach to extract grammar for existing languages. They recover grammar from language references, compilers, and other artifacts. This paper is a follow up to the authors earlier work mentioned in the previous section. This paper demonstrates a structured process to obtain a grammar specification and to derive a realistic parser from it. They have used several grammar artifacts such as compilers, debuggers, animators, profilers, pretty printers, language reference manuals, language browsers, software analysis tools, code preprocessing tools, software modification tools, test-set generation tools, software testing tools, etc. Thus the authors use this whole gamut of grammar source to recover grammar of the language. The process involves extraction from IBMs Reference Manual, connecting the raw grammar (i.e. taking care of undefined and unreachable nonterminals), test driving the grammar by generating parser from it and ensuring quality of the grammar.

Reverse Engineering the Compiler

When a compiler is designed carefully with high quality of code ,it is possible to extract grammar of the language from the source code. In [20] Sellink and Verhoef discuss an approach to recover the grammar using compiler source code. This work is also related to the previous two works.

The first step in the process is to detect the portion of the compiler which contains the source code for the parser. For generating parsers, compiler writers usually use automatic parser generators like Yacc [7], which generate parsers automatically from the language description called parser specifications. In this case, the source files that are responsible for parser can be located by searching for keywords of the language or just by looking for files that end with `.y`. These source files contain the grammar of the language in the form of parser specifications. Besides the grammar the files also contain some additional code called semantic actions for compilation purposes like generating syntax tree etc. This additional code is removed from the parser specifications, as the parser specifications are usually written in BNF-like notation, obtaining the grammar from the specifications is trivial. Once the grammar is obtained tools described in the [19] are used to improve its quality.

The legacy applications are written in languages that are old and often unsupported, so the source code for their compilers is not available. As a result, the approach discussed above can not be used to extract the grammar.

1.3.2 Modifiable Grammars

Some context sensitive constructs can be handled by using *modifiable grammars* that handle static semantics of a language through dynamic context free syntax rules. Two approaches that use modifiable grammar are mentioned : Universal Syntax and Semantics Analyzer and Dynamic Parsers.

Universal Syntax and Semantics Analyzer

In [2] Burshteyn presents the design of a tool called *Universal Syntax and Semantics Analyzer* (USSA) for defining syntax and semantics of formal systems. A modifiable grammar allows for the introduction of new syntax rules and the deletion of existing syntax rules during the generation or parsing of sentences.

The USSA takes a language definition in the form of a modifiable grammar. The definition is written in a *Yacc* like notation, consisting of two sections called the *modification block* and the *no-modification block*. The parsing of a sentence begins with an initial set of rules called the *current set* obtained from the *no-modification block*.[2] While parsing a sentence, USSA applies rules from the current set and also modifies the current set simultaneously by adding and deleting rules specified in the modification block. This process continues until the sentence is successfully parsed, the grammar can now be obtained from the current set of rules.

As discussed above, USSA works by taking two set of rules, the initial set and additional rules in the modification block that can be added or deleted from the current set as the parsing continues. So, USSA basically works to-wards generating a program specific grammar from a generalized grammar. In the problem at hand, we only have the approximate grammar which is incomplete, the missing rules are not available to us in the form of modification blocks, so this approach can not be used for obtaining the grammar.

[2]The same process is followed for the generation of sentences.

Dynamic Parsers and Evolving Grammars

Cabasino *et al* present the concepts of evolving grammars and dynamic parsers in [17]. Evolving grammars are defined as a succession of static grammars that evolve at parse time. Dynamic parsers follow the evolution of grammar during source program parsing and allow a syntactic-only parsing of programs written in problem adaptable programming languages. This enables the user to introduce new operators, statements and data types in a program.

A grammar evolution starts with an initial grammar G_0 and continues to G_n as a succession of grammars G_1, G_2, ... connected by production rules. These production rules change the grammar from G_i to G_{i+1} on a particular reduction. So, the parsing begins with an initial grammar and as it continues new production rules specific for the source program being parsed are added in the growing grammar. As a result, the grammar evolves towards a program specific context free grammar. For working with evolving grammars, one needs a parser that can adapt itself to a new grammar during source program parsing. A dynamic parser does so by changing its parsing tables at parse time while preserving the symbols present on the stack.

Similarly to the previous approach, this approach is also based on the assumption that the grammar for the program is available but as a succession of static grammars. The approach then works to-wards obtaining a program specific grammar. So, in the case when these grammars are not available, the approach can not be used for obtaining the grammar.

1.3.3 Natural Language Processing Approaches

In the realm of natural language processing, the problems with large complex grammars are also well known. In [12] we can read that formal grammars for natural languages can become unmanageable when they evolve. It is also noted that grammars are usually written by a small group of people. These people are the only ones who are able to understand its structure. If they leave, the grammar cannot be maintained anymore since its size and complexity prevent anyone else from gaining insight in it. Therefore, in [12] tool support is provided for affix grammars over a finite lattice(AGFL) for natural language. An affix grammar can be seen as a context-free

grammar in which the productions are extended with *affixes* (cf. parameters, attributes, or features) to express agreement between the lhs and the rhs, and between the members of the rhs.

The tool support consists of modularity support, analysis support for grammars, random test sentence generation, and a parser generator with trace facilities. The tool support was developed in the natural language front end project aiming at the construction of comprehensive and well-validated grammars and efficient parsers for a number of European Languages. So their focus is on development of grammars for natural languages. Our focus is on reengineering of existing grammars for computer languages.

1.3.4 Grammar Inference

In [13] the authors have provided several approaches to learning syntax of a language from a set of correct and incorrect sentences. The exact learning of the context free grammars is made difficult by the fact that several decision problems related to them are undecidable. The authors discuss several heuristic approaches for learning approximations to the target grammar.

An approach called *"Learning Recursive Transition Networks"* is discussed that works by converting grammatically correct sentences into transition networks that are similar to finite state diagrams. Similar networks obtained from different sentences are then merged repeatedly to give more general transition networks. The language grammar can be obtained from these networks.

Another approach called *"Learning CFG using Version Spaces"* works in the following way. From a set of positive examples, a trivial grammar is constructed that accepts exactly the set of these examples. A *version space* is created now which is defined as the set of all possible generalizations of the trivial grammar that are consistent with the examples. More examples are processed now, each positive example results in the expansion of the version space by considering the grammar for the example and each negative example results in the reduction of the version space by eliminating the grammars that accept the example. The process is repeated for all the

examples at hand, and a version space for these examples is obtained. The grammar can now be obtained by combining the grammars present in the version space.

The authors also discuss several other approaches like *"Learning NPDA using Genetic Search"* and *"Learning Deterministic CFG using Connectionist Networks"* are also discussed.

1.3.5 Genetic Algorithm Approach

In [1] Dario Bianchi proposes a credit assignment algorithm for inferring context free grammar from a set of sample sentences. Grammar is represented as a population of rules. A strength, associated to each rule, takes into account its usefulness in correctly classifying the test sentences. This method also relies on availability of negative sentences. Moreover the rules must be known beforehand. The algorithm chooses the rules based on a fitness value assigned to it depending on how many times each rule is fired while recognizing a valid sentence and an invalid sentence .i.e. sentence not belonging to the target language .

The problem with the approaches discussed is that they rely on the availability of both positive and negative examples of sentences in the languages, which makes them difficult to use. Also, as these approaches are heuristic in nature they perform well on small to medium sized problems but do not scale well to larger problem sizes.

1.4 Scope of Our Work

With the relevance of grammar in program analysis tools well established, it is clear that a tool to extract grammar from source programs will be very useful. We have proposed an engineering approach whereby we attempt to extract grammar from the source code of the program. In [6] a new approach to recovering grammar from programs was proposed. We have enhanced the approach further and developed searching algorithms. We have tested our method with C and C++ grammars and developed our algorithms accordingly. The algorithms that use these heuristics have been integrated into a tool called *GramEx*.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows.

- In chapter 2, we will discuss our approach of extracting grammar from programs. We follow the same interactive and iterative approach for grammar extraction. This chapter focuses on algorithms and heuristics for searching for missing rules.
- In chapter 3, we will discuss implementation details and features of our tool *GramEx* which is based on the techniques discussed in chapter2.
- In Chapter 4, we give our experience of working with the tool. We also present our experiments and finally conclusion.

Chapter 2

Grammar Extraction from Programs

2.1 Context Free Grammars and their Dialects

Context-free grammar (CFG), which is also sometimes called BNF (Backus-Naur Form), is a notation which is used for the syntactic specification of a programming language. A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors. The advantage of using CFG is that, an efficient parser can be constructed automatically from a properly designed grammar. There are a few constructs found in some programming languages which cannot be described using context free grammars. But such constructs are rare, and CFGs are useful for describing all the constructs of most modern programming languages. Mostly the tools that generate parsers automatically, like *Yacc*, make use of CFGs. Parsers for most modern programming languages like for C, C++, Java or COBOL use context free grammars.

Thus all high level declarative programming languages are defined by a CFG. It is natural to define certain programming language constructs recursively. For example all C programs are composed of a translation unit. A translation unit in turn is made of function declarations and definitions and variable definitions. A function may contain blocks of statements and so on. This kind of recursive definition is captured in BNF as follows.

```
<translation_unit>      --> <external_declaration> |
                            <translation_unit> <external_declaration>
<external_declaration>  --> <function_definition> | <variable_declaration>
<variable_declaration>  --> <type_name> IDENTIFIER ';'
<function_definition>   --> <type_name> <name_of_function><block>
<type_name>             --> INT | FLOAT | BOOLEAN
<block>                 --> '{' <statement_list> '}'
<statement_list>        --> <statement> |     <statement><statement_list>
<statement>             --> IF <expression> THEN <statement>
```

Figure 2.1: BNF notation

Figure 2.1 shows a BNF notation for a subset of C programming language. This CFG describes grammar for a dialect of C. We can go on adding rules to this notation and enhance the grammar so that the programming language becomes richer and supports more constructs. For example this language supports the *IF* statement. We can enrich the language by adding the following rule to the grammar.

```
<statement> --> IF <expression> THEN <statement> ELSE <statement>
```

Now this language supports the *IF-ELSE* construct also. Suppose we have access only to the first grammar but programs written in this dialect of C also use the *IF-ELSE* construct. Hence the rule for *IF-ELSE* is missing from our grammar. Our tool intends to find such missing rules and incorporate them into the grammar to make it complete. This is not a trivial task as we have to take care of various issues. Some of the issues are how to find the rules that would fit correctly in the incomplete grammar. Even if we find the correct rule it can only be inserted in the grammar after the correct nonterminals symbols are found for the rule.

One of the main reasons for creation of a dialect is the non-availability of some useful constructs in its parent language. That is, the parent language is viewed as lacking some constructs that are probably available in some other language. Hence to enrich the language, these constructs may be added in the language, giving rise to a dialect. We define a dialect of a language as a language that has some constructs that the parent language does not have. The overall structure or the grammar of a dialect is not very different from its parent. For the purpose of grammar extraction

we are concerned only with new constructs. Some constructs that may be deleted do not concern us as they are essentially the same as the constructs which are there in the language but not used in the programs.

For any Language L there are several grammars G_1, G_2, .. , G_n that can generate the language. Even the program itself is the grammar of the language. So a trivial solution to this problem could have been to include all the lines that give parsing error as a rule in the grammar, each with a new nonterminal as its l.h.s. But this grammar would not be able to parse any other program, even of the same language. The grammar would also be of not much use because it will not capture the syntactic structure of the programs to any degree. Our ultimate aim is to automatically construct some grammar based tools like *Test Coverage Analyzer* from the grammar. These tools are constructed by instrumenting the grammar file. Such a grammar will be of no use to us for reengineering purposes.

2.2 Extracting Context Free Grammar

The context free grammar extraction problem is as follows : Given a finite set of positive examples (sentences belonging to the language of the target grammar) and an approximate grammar $\bar{G}$ identify a context free grammar G which is equivalent[1] to the target grammar G_t. The target grammar is a dialect of the initial approximate grammar $\bar{G}$. We have a grammar available for the language which forms the approximate grammar for the dialect of the language. We also have a set of programs that belong to this language. We want to obtain the complete grammar for this dialect with minimum inputs from the user. Ours is a problem of Grammar Recovery or Grammar Extraction. It is an engineering approach where we locate the deficiency in the grammar and then search for the missing rule in a database of rules that has been constructed beforehand. It *is not* a problem of Grammar Learning or Grammar Inference. We are *not* attempting to learn new rules from studying the sample sentences. Rather we are only attempting to *find* the correct rules for the set of programs that we started off with.

[1] Two grammars G_1 and G_2 are equivalent if the language generated by both of them is exactly the same.

Grammar Learning approaches have several issues which make them unsuitable for our purpose. There are several machine learning and AI approaches for learning grammar. Inducing grammar from programs is not a trivial problem. The theoretical limitations that plague regular grammar inference are also present in context free case because the set of context free languages properly contains the set of regular language. The problems are further compounded by the fact that several decision problems related to CFG are undecidable. Given two CFG G_1 and G_2 there exists no algorithm that can determine whether G_1 is *more general than* G_2 (i.e., $L(G_1) \supseteq L(G_2)$). Similarly, there exists no algorithm that that can answer the question "Is $L(G_1) \cap L(G_2) = \emptyset$?" Thus, even in learning of the target grammar exact learning is seldom attempted in the case of CFG. Only heuristic approaches for learning the grammar exist.

As mentioned before, for a language L, there exist several grammars G_1,G_2,..G_n which generate the same L. Gold proved that regular grammars cannot be learned from positive examples alone [4]. This also holds for Context Free Languages [13]. So we need both positive and negative examples to recover the complete grammar. But in our scenario we have access only to positive examples i.e. syntactically correct programs.

We also have access to an initial or approximate grammar. Without the initial grammar it is very difficult just to find when one construct ends and another starts. This problem becomes more significant when one construct contains another construct, for example the body of a *for* loop contains an *if* statement. If an approximate grammar for the dialect is available, which may be the grammar of the parent language, the problem becomes simpler. In this case, we need to extract grammar only for the constructs, which are new in the dialect. Since we are dealing only with formal languages syntax is the only concern for us not the semantics. Programming languages also have well defined structure and thus we do not use Natural Language Processing and Machine Learning techniques for this problem.

2.2.1 Exploiting Commonalities in Grammar

The class of programming languages that we are dealing with (declarative, block structured) like C, C++, Pascal and Java have many syntactic structures that are

quite similar. For example all of them have similar loop structures, decision structures like *if-else* constructs and *switch-case* constructs, etc. So it is not very unlikely that dialects of these languages also will use similar structures and blocks. So we can construct a *metagrammar* that captures these common constructs among different languages. A metagrammar is an abstraction of different grammars. We can construct a set of *generic rules or rule templates* which after some modification if included in the grammar will be able to parse the new constructs. We call such a collection of generic rules *Knowledge Base*. Such a repository of generic rules is critical in our scheme of recovering grammar. Ultimately we feel that this Knowledge Base will be complete and comprehensive in the sense that it will contain most constructs of modern programming languages. So the rule for most of the new constructs can be found in this Knowledge Base. So our problem of grammar recovery reduces to one of finding the missing rule from this Knowledge Base and plugging it in the grammar at the proper place.

There are two approaches in which we can perform grammar extraction from an initial grammar.

- **Top Down Approach:** In this approach we can start with a minimal set of rules as our initial grammar. This grammar is incomplete. We then create a parser from this grammar and try to parse the programs in our collection. Since the grammar is incomplete there will be parsing errors. The Knowledge Base is searched for a matching rule, the rule is added, the parser is reconstructed and the entire process is repeated. This continues till the entire grammar is recovered.

- **Bottom Up Approach:** We can start with an exhaustive grammar which contains all the rules of the Knowledge Base. We then parse all the programs and note the rules that get fired. We eliminate those rules that are not used. Thus we obtain the minimal grammar.

The Top Down approach depends on existence of a predefined set of rules which we call the Knowledge Base. This database is consulted by the program every time a rule is missing and parsing cannot proceed further. The Knowledge Base is a very important component of this scheme. But it is conceivable that sometimes the Knowledge Base will not be able to throw up any matching rule. In such a case

the user will have to intervene and create a new rule. The Knowledge Base is also updated to include the new rule template.

The Bottom Up approach on the other hand starts of with a complete grammar and then the grammar is made smaller by eliminating unused rules. It is difficult if not impossible to come up with such a complete grammar in the first place. We therefore use the Top Down Approach in our technique.

Automating the entire process of grammar extraction is difficult. There are several issues that needs to be taken care of. First of all it is difficult to create an exhaustive list of rules which will parse all programs of any given language. There is also the important issue of finding the correct rule that will be able to resolve the parsing error. There may be multiple matches. So this rule matching involves searching through the Knowledge Base with certain closeness metric which will prioritize the rules. Then we also need to take cognizance of the fact symbols in the approximate grammar are likely to be different than symbols in the generic rules. Moreover, locating the correct position in the grammar to put the rule is also a non-trivial task. Therefore, any system cannot be expected to recover the grammar automatically from the source program. Instead, obtaining the grammar should be viewed as a cooperative process involving both the system and the user. The system finds the deficiencies in the grammar and possible solutions. The user assists it in making decisions for choosing a rule from many probable rules. The user also helps in modifying the rule so that it contains only the symbols that are defined in the grammar.

After adding the rule and all its corresponding nonterminals, the grammar may be modified at more than one place. This may give rise to some rule conflicts. This means that the newly added rule may result in some constructs not being parsed properly. To correct this user may need to repeat the process. So the final correct grammar will be typically obtained after multiple iterations. After every new rule is added parsing must again begin from the start of the source file.

2.3 Our Approach To Grammar Extraction

An Interactive, Iterative Approach: Our approach is an iterative and interactive one where the tool creates a parser from an initial grammar. The initial grammar is the grammar of the standard language. Then our tool parses the source program, locates the construct which cannot be parsed and gives suggestions to the user which rule should be included in the grammar. There may be multiple rules returned as possible rules. We have designed an algorithm to prioritize the rules based on a closeness metric. The ultimate decision on which rule to chose is left to the user. The user can select one of the shortlisted rules(in case of multiple match) and modify its nonterminals or simply modify the matched rule(in case of single match) or create a completely new rule(in case of no match). The system adds this rule to the yacc file and updates the parser. The user then resumes parsing process with this new parser. This activity is repeated until the whole grammar is recovered. Figure 2.2 gives an overview of our approach. As can be seen this a modification of the Top Down Approach that we had looked at earlier.

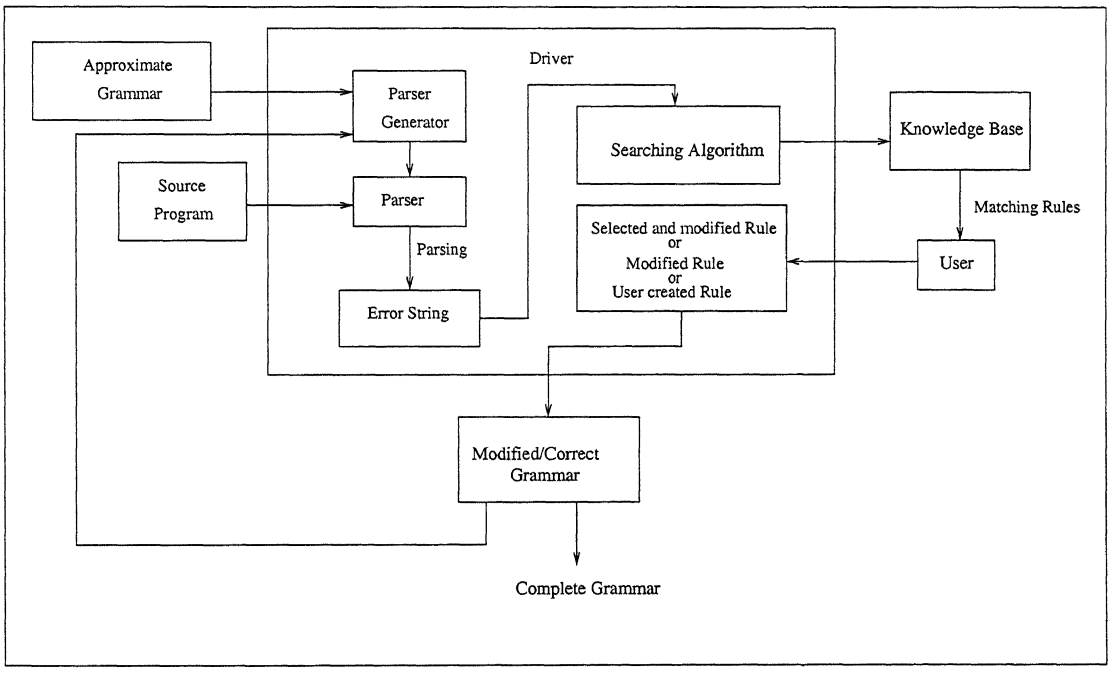


Figure 2.2: Overview of our Semi Automatic Grammar Recovery Approach

The design requires an approximate grammar and source programs as input. The

Knowledge Base is the other input to the tool. It creates the parser from the approximate grammar and parses the source program. When an error string is obtained it is sent to the searching algorithm to select the matching rules from the Knowledge Base. If there are no matching rules found then the user has to create a new rule. The new rule is added to the approximate grammar to get the modified grammar. The parser is reconstructed from the new grammar and parsing begins again on the source program from the beginning. This process is repeated till the entire program is parsed successfully. We will describe all the steps involved in the process in detail in the following sections.

2.3.1 Parsing the Source Program

This is the first step that user takes. We have an initial approximate grammar. We create the parser from this grammar using `yacc` [7]. `Yacc` is a widely used tool to generate parsers automatically from the BNF notation. Yacc produces a shift-reduce LALR(1) parser. It turns the specification file into a C program, which parses the input according to the specifications given. This parser is used to parse the source program iteratively.

The parser produced by `yacc` consists of a finite-state machine with a stack of states. The machine has only four actions available to it, called *shift, reduce, accept and error.*

- The *shift* action is the most common action taken. Consider a parser stack having 56 as the current state on top of stack. In state 56 there may be an action to shift state 34 if the lookahead token is IF. Whenever a shift action is taken, there is always a lookahead token. Thus 34 becomes current state (on top of stack) when IF is seen at the input.

- Reduce action is appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. The lookahead token may be consulted to decide whether to reduce.
  Suppose the rule being reduced is :
  `A : x y z ;`
  Then the actions involved will be to pop the three states from too of stack.

These states were pushed while recognizing `x,y` and `z`. Then shift the state corresponding to `A`. To differentiate from shift of a token this action is called `goto` action.

- The `accept` action indicates that the entire input has been seen and it matches the specification. It is taken when the endmarker is seen.
- The `error` action represents a place where the parser can no longer continue parsing according to the specification. The input token it has seen, together with the lookahead token cannot be followed by anything that would result in a legal input.

Modifications Done to the Yacc Code: We have enhanced the `yacc` source code slightly to obtain more information at the point the parsers encounters an error. This not only helps in determining the line number of the new construct, but potentially also the structure of the new construct. The `yacc` code contains two stacks: one for storing the parser states and one for storing values returned from the lexical analyzer and the actions. We have added a third stack which will contain the symbols from the input file i.e. terminals and nonterminals. *We have not touched existing yacc code to rule out the possibility of introducing any bug in the tried and tested yacc code.* Thus our stack runs in parallel with the `yacc` stacks and saves all the unreduced terminals and the nonterminals which results from a reduction. Every time a shift occurs the new terminal is pushed on to this auxiliary stack and every time a reduce occurs, the number of symbols equal to the right hand side of the corresponding nonterminal are popped and the new nonterminal is pushed on the auxiliary stack.

When an error occurs we continue scanning input from the source file (by calling the lexical analyzer) till we get a synchronization symbol (chosen currently as semicolon). We keep pushing these symbols onto the stack. From this stack, we constitute the error string as follows. We start from the bottom of the stack and move upwards till we hit the first terminal. We create the error string as a concatenation of this terminal and all the symbols that we encounter till the top of the stack. This error string can later be used to help determine the missing grammar rule. The reason why we look for the first terminal is that everything before that is the left context of the construct which causes error. The parser hit the error state somewhere after this terminal. So

it was not reduced and replaced by a nonterminal. The error string ideally should contain all the keywords of the new construct so that the correct missing rule can be found.

The algorithm used to obtain the error string is shown in Fig. 2.3 . We need to find the first terminal from the bottom of the stack. We set the string variable P to bottom of the stack (in line 2) and move up the stack till we reach the first terminal (in lines 3-4). The Error_Str is created by concatenating all symbols, including this terminal, till the top of stack (in lines 5-10). Thus Error_Str is composed of symbols which are formed by looking down into the parser stack as well as looking ahead in the input file, from the point that yacc actually goes into error state.

```
Algorithm Error_String(St,V_T)
Inputs
        St : Auxiliary Stack Obtained by modifying yacc code;
        V_T : Set of Terminals of the Grammar;
Output
        Error_Str : String which will be obtained by concatenating the symbols of S;
Declare
        P : String variable ;
        TOP : Top of stack;
Begin
[1]     Error_Str = ∅ ;
[2]     P = S[0]; /* Set P to bottom of stack S */
[3]     while(P not ∈ V_T )
[4]                 P = S[I++]; /* move up the stack*/
[5]     repeat
[6]             Begin
[7]                 Error_Str = Error_Str + P ; /* concatenate stack symbols*/
[8]                 P = S[I++]; /* Move up on the stack*/
[9]             End
[10]    until(I ≠ TOP )
End
```

Figure 2.3: Algorithm For Obtaining Error String From Parser Stack

The Knowledge Base

In this section we provide the description of the Knowledge Base. As briefly mentioned earlier the Knowledge Base is a repository of generic rules. It is searched whenever an error occurs while parsing the source file. A record of the Knowledge Base contains three fields: a keyword for the rule, a category for the rule and the rule itself.

Rule No	Keyword	Category	Generic Rule
1	if	selection	<selection_statement> : IF '(' <expression> ')' <statement>
2	else	selection	<selection_statement> : IF '(' <expression> ')' <statement> ELSE <statement>
3	switch	selection	<selection_statement> : SWITCH '(' <expression> ')' <statement>
4	while	iteration	<iteration_statement> : WHILE '(' <expression_opt> ')' <statement>
5	do	iteration	<iteration_statement> : DO <statement> WHILE '(' <expression> ')' ';'
6	for	iteration	<iteration_statement> : FOR '(' <expression_opt> ';'<expression_opt> ';'<expression_opt> ')' <statement>
7	for	iteration	<iteration_statement> : FOR '(' <declaration> <expression_opt> ';'<expression_opt> ';'<expression_opt> ')' <statement>
8	repeat	iteration	<iteration_statement> : REPEAT '(' <statement> ')' UNTIL <expression>
9	case	statement	<labeled_statement> : CASE <constant_expression> ':' <statement>
10	default	statement	<labeled_statement> : DEFAULT ':' <statement>
11	break	statement	<jump_statement> : BREAK ';'
12	continue	statement	<jump_statement> : CONTINUE ';'
13	goto	statement	<jump_statement> : GOTO <identifier> ';'

Table 2.1: Samples of Generic Rules from Database

A snapshot of the Knowledge Base is shown in Table. 2.1. The rules in the Knowledge Base, are in BNF. We have enclosed the nonterminals in angular brackets (< >). The nonterminals may be different in the grammar files written by different people. For example someone can use *stmt* instead of *statement* while writing the yacc source file. Therefore, before inserting any generic rule into the grammar file we need to take care of this aspect. So the generic rule needs to be made grammar specific by replacing the nonterminals by the actual nonterminals used in the grammar. The replacement string should be provided by the grammar writer. One approach is to provide a mapping between the nonterminals used in approximate grammar and those used by the Knowledge Base. Such mappings are easy to create.

2.3.2 Searching for Missing Rules in the Knowledge Base

If the grammar is incomplete then the parser will encounter an error. We now need to find the rule which when included in the grammar will make it complete. Our modified parsing technique allows us to obtain the error string which contains the missing construct. With this string we search the Knowledge Base. We will describe a couple of heuristics that we have developed and employed to search the Knowledge Base.

Simple Heuristic

The string we use for searching for the missing rule is obtained by the Error_String algorithm shown in [Fig.2.3]. The algorithm for matching rules is called Match_Rule and it works as follows. From the error string we form a set of symbols. We take each rule from the Knowledge Base one by one and again form a set of keywords which are used in that particular rule. All the rules which have some keywords in common to the set formed from error string are selected. All these rules are made available to the user, who can pick up the appropriate rule. The rules are not presented in any particular order.

The algorithm is shown in Fig.2.4. The set of selected rules is initially empty (line 2). Then each rule in the Knowledge Base is examined one by one and set of keyword is constructed for each rule (line 3-9). Then we perform an intersection over this set and set of keywords constructed from Error_Str. If the resulting set is non-null we

```
Algorithm Match_Rule(Str,KB)
Inputs
      Str: Error String obtained from algorithm 2.3;
      KB : Set of generic rules; /*present in Knowledge Base*/
Output
      SelectedRules : Set of Generic Rules that match Str;
Declare
      Error_Set : Set of keywords obtained from Str ;
      Rule_Set : Set of keywords obtained from a Generic Rule ;
Begin
[1]   Error_Set = {keywords in Str } ;
[2]   SelectedRules = ∅;
[3]   for(i=1 to number of rules in KB )
[4]         Begin
[5]               read rule(i);
[6]               Rule_Set = {keywords in rule(i)} ;
[7]               if( (Rule_Set ∩ Error_Set) ≠ ∅)
[8]                     SelectedRules = SelectedRules ∪ {rule(i)} ;
[9]         End
End
```

Figure 2.4: Algorithm for matching generic rules from Knowledge Base

include that rule in SelectedRules set. All the selected rules are shown to the user.

Heuristics Used to rank rules

In this subsection we present another heuristic which we have used to rank all the selected rules in case multiple matches are found. This heuristic is based on a closeness metric that we have defined. Let us suppose that we want to evaluate closeness of a rule *rule(i)* and the error string *Error_Str*. We first extract all the keywords from the Error_String and rule(i) and form two *ordered* sets S_1 and S_2 respectively. By ordered set we mean that while looking at Error_Str and rule(i) we include the keywords in the set in the same order that we encounter them in these strings. For e.g. if the Error_Str is

```
IF '(' IDENTIFIER '<' INTEGERconstant ')' THEN IDENTIFIER '='
INTEGERconstant '*' IDENTIFIER ';'
```

and the rule string is

```
selection_statement :  IF '(' <bool_expr> ')' THEN <statement> ELSE
<statement>
```

then $S_1 = \{\text{IF , THEN}\}$
and $S_2 = \{\text{IF , THEN , ELSE }\}$.

We then use a closeness metric to find proximity between pairs of such ordered sets and assign ranks to the rules. We present the algorithm for computing closeness Compute_Proximity in Fig. 2.5 and 2.6. It computes the closeness of a rule with the error string and returns the value in the variable Proximity. The variables α and δ compute the positions of the first matching keyword in the RS and ES respectively. They are initialized to 1. The variable β is used to assign the proximity to the rule. The final value of proximity is computed in several steps and depends on three factors: The number of matching keywords, their relative position in the keyword sets and number of unmatched keywords.

The proximity of a rule is calculated in a weighted manner as follows. The highest weightage is assigned to the first matching keyword. This value is chosen as 4. The value of parameter β was arrived at after analyzing two C++ grammars. It was found that that there was no rule which had more than 2 keywords in it. Thus it is highly unlikely that any rule that is introduced in the grammar will have 3 keywords. We have decided that 4 should be the value of β i.e. one higher than the case where three keywords are found. The value of 4 is a conservative estimate on the weightage. We have also found through our experiments that seldom more than 2 keywords match. This value of β works in the experiments that we have performed. Experiments are covered in chapter 4. To obtain the first matching keyword we need to obtain the first element in common to both the sets. So we skip all the symbols from RS and ES before that (in lines 4-21). The elements are removed from the two sets after they are picked up. After the first match occurs the rule is assigned proximity of 4 (line 21).

We then compare the remaining elements in the two sets one by one and assign higher proximity to that rule in which more keywords match. This proximity is

assigned as follows. A match at the second place results in more weightage than a match in the third place and so on. So β is decremented in every iteration (line 29). The final value of proximity also takes into account values of α and δ. This means if the first keyword match occurs in i^{th} position in both RS and ES then the proximity will be higher. Also if there is any extra, unmatched element we reduce the proximity by 1 for each extra keyword. This is computed by the expression in line 33, where d is assigned the difference in the number of elements of ES and RS. Thus if there are any extra keywords *after* the last match then d has a nonzero value. Note, since we removed elements from the two sets after picking them in lines 8 and 12, d will always contain the number of unmatched keywords after at least one match has occurred. The final value of proximity takes into account the relative position of the first match as well as the extra unmatched keyword. The expression $|\alpha - \delta|$ computes the difference between the position of first common keyword in the two sets RS and ES. The more this difference the lesser is the proximity value. The extra unmatched keyword is taken care of in the term | d |.

Let us take an example and run through the algorithm. Consider
ES = { IF,THEN }
RS = {IF,THEN,ELSE}
$\alpha = 1; \delta = 1;$/*the common keyword IF occurs in the first position in the two sets.*/
P = 4; in line 21
P = 4 + 3 = 7; in line 30
d = 1; in line 33 /* after the last match THEN there are 0 keywords in ES and 1 keyword in RS. */
P = 7 - 0 - 1 = 6;in line 35
So the rule gets a proximity value of 6.

Consider another example
ES = { IF,THEN }
RS = {IF,THEN}
$\alpha = 1; \delta = 1;$
P = 4;
P = 4 + 3 = 7;

d = 0;
P = 7 - 0 - 0 = 7;
So the rule gets a proximity value of 7 and is thus ranked higher than the previous rule.

Based on this ranking heuristic we designed another matching algorithm which now prioritizes the rules short listed from the Knowledge Base. The new algorithm takes as input the selected rules which is the output of MatchRule algorithm 2.4 and then ranks them according to the algorithm 2.5 and 2.6. The output is a list of rules which are arranged according to a priority. This algorithm is shown in Fig.2.7.

Second Level Rules

If the new rule introduces some new nonterminal which has not yet been defined in the grammar, then a rule needs to be given for that nonterminal also. Such a rule is called a *second level* rule. In such a case our tool makes suggestions as to what this rule will be.

Thus a second level rule is a rule where there are one or more nonterminals which are not defined in the grammar. More than one rule needs to be added to the grammar for it to be complete and well-defined. This is because all the undefined nonterminals also needs to be taken care of. We use a simple method to guess second level rules. This follows from the rule ranking algorithm that is shown in Fig.2.5 and 2.6. In Rank Rule algorithm the closest matching rule for an error string is found out by matching the keywords in the rule and the error string. The strings between the matching keywords in the rule are nonterminals of the grammar and the strings between keywords in the error strings are terminal symbols. We construct a second level rule by setting the nonterminal in l.h.s and the terminals in the r.h.s. We will provide an example to illustrate it further. Let us consider the previous example. The missing rule was

```
iteration_statement :  WHILE '(' expression ')' statement
```

and the error string was

```
WHILE '(' IDENTIFIER '<' INTEGERconstant ')' IDENTIFIER ICR ';'
```

If the grammar file does not contain definition for the nonterminals `expression` and `statement` then our algorithm will make the following rules as suggestion :

1. `expression : IDENTIFIER '<' INTEGERconstant`

2. `statement : IDENTIFIER ICR ';'`

This is because here the keyword `WHILE` has matched. So the remaining string between the brackets will be an `expression`.

Rules added by User

The Knowledge Base grows with more use of the tool. It is not exhaustive. So it is possible that no matching rule is found in the Knowledge Base and the user has to create a new rule. Our tool provides a GUI for doing this. Once the user writes a new rule user can also include a rule template in the Knowledge Base. Thus the next the time we use this tool and such a construct is encountered while parsing we can find the rule in our Knowledge Base. The user can also search the Knowledge base using keyword search or using category search. For e.g. the rule for `for` loop can be searched either by the keyword *for* or by the category iteration.

2.3.3 Modifying the Grammar

The new rule found from the database or created by the user now is added to the grammar. All the newly introduced symbols from the rule must be defined. Our tool gives a list of all the new nonterminals and terminals currently undefined to help the user. If a new terminal is added then the lexical analyzer needs to change too. The tool provides interface for making these changes. The new rule must be added at the correct place in the grammar. The tool searches the grammar for a rule with the same l.h.s as this rule. If found it adds the rule next to it. Otherwise the new rule is added at the end.

The parser is reconstructed from the new, modified grammar. Parsing of the source program begins from the first line. This entire process is thus repeated until the program is parsed correctly and there are no more syntax errors.

2.4 Summary

In this chapter we have described our approach to Grammar Extraction. It is an interactive and iterative approach. As the new constructs found in a dialect are usually available in some other languages, the probability of their being present in the Knowledge Base is quite high. The Knowledge Base used by grows with more use. As it grows in size the need for user to create a rule also decreases because chances are high that the rule will be found in the Knowledge Base. We have proposed and implemented some closeness metric while searching for the missing rule in the Knowledge Base.

Our approach is quite simple and heuristics provided are easy to understand and implement. No complex datastructure is required to implement it.

```
Algorithm Compute_Proximity(RS,ES)
Inputs
    RS : Set of Keywords in Rule;
    ES : Set of Keywords in Error_Str;
Output
    Proximity: Number indicating closeness of the two sets;
    α,β,δ,d : Variables of type Integer;
    matched : boolean;
Declare
    e_R : Element of set RS ;
    e_E : Element of set ES ;
Begin
[1] β = 4; /*Used to calculate Proximity of rule*/
[2] α = 1; δ = 1; /*Used to find relative position of first matching keyword*/
[3] while (more elements in ES)
[4]     Begin
[5]         matched = false; /* Boolean variable */
[6]         RS = Set of Keywords in Rule;
[7]         Pick first element e_E of ES;
[8]         Remove e_E from ES;
[9]         while (more elements in RS)
[10]            Begin
[11]                Pick first element e_R from RS
[12]                Remove e_R from RS;
[13]                if (e_R == e_E) /* Equality is string comparison */
[14]                Begin
[15]                    matched=true;
[16]                    break;
[17]                End
[18]                else α = α + 1;
            /* α counts the position of matching keyword in RS */
[15]            End
[16]            if (matched == true) break;
[18]            else δ = δ + 1;
            /* δ counts the position of matching keyword in ES */
[20]    End
            /* At this point the first match in the two keyword sets has been found */
[21]    Priority = β; /* Priority is highest for first match */
```

Figure 2.5: Algorithm Compute Proximity (continued next page)

```
[22]  while(more elements in ES )
[23]        Begin
[24]              Pick e_R from RS;
[25]              Pick e_E from ES;
[26]              if ( e_E == e_R )
[27]              if(β > 1)
[28]                    Begin
[29]                          β = β − 1;
[30]                          Proximity = Proximity + β ;
[31]                    End
[32]        End
[33]        d = number of elements in ES
[34]                          − number of elements in RS ;
[35]        Proximity = Proximity − |α − δ| − | d |;
End
```

Figure 2.6: Algorithm Compute Proximity

```
Algorithm Rank_Rule(Str,SelectedRules)
Inputs
      Str: Error String obtained from algorithm 2.3;
      SelectedRules : Set of rules which are selected by algorithm MatchRule;
Output
      PrioritizedRules : Set of Prioritized Rules;
Declare
      Error_Set : Set of keywords obtained from Str ;
      Rule_Set : Set of keywords obtained from the Generic Rule ;
      Priority : Rank assigned to a rule;
Begin
[1]   Error_Set = {keywords in Str } ;
[3]   while(more rules in SelectedRules )
[4]         Begin
[5]               read rule(i);
[6]               Rule_Set = {keywords in rule(i)} ;
[7]               Priority = Compute_Proximity(Rule_Set,Error_Set);
[9]               PrioritizedRules = PrioritizeRules ∪ {(rule(i),Priority)} ;
[10]        End
[11]  Sort PrioritizedRules by Priority;
End
```

Figure 2.7: Modified Algorithm For matching generic rules

Chapter 3

Implementation

We have implemented a GUI based tool for extracting grammar called *GramEx*. The tool incorporates the algorithms for searching the Knowledge Base that we have discussed previously. The tool has been written using Java and the Lex and Yacc tools. Some of the main features of the tool are as follows:

- At the start the tool asks the user for the source program, the Lex and Yacc specifications of the grammar, and the mapping of the nonterminals used in the approximate grammar and those used in the Knowledge Base .
- On encountering an "error", it shows the source code with the error portion highlighted. It also gives all the matching rules and allows the user to select any of these.
- If no matching rules exist in the Knowledge Base then the user is provided with a GUI to create a new rule.
- A separate window is provided for browsing the Knowledge Base.
- The system allows user to undo changes made to the grammar. That is, if after some rule has been added , if the user does not find the modification satisfactory, he can revert back to the grammar before this addition was made.
- Once one program has been successfully parsed (with the new grammar), the user can specify the next file to be parsed.

3.1 Main Components of GramEx

In this section, we will discuss the design of the tool in brief. The architecture of the system is shown in Figure 3.1. As shown in the figure, the system consists of four main components:

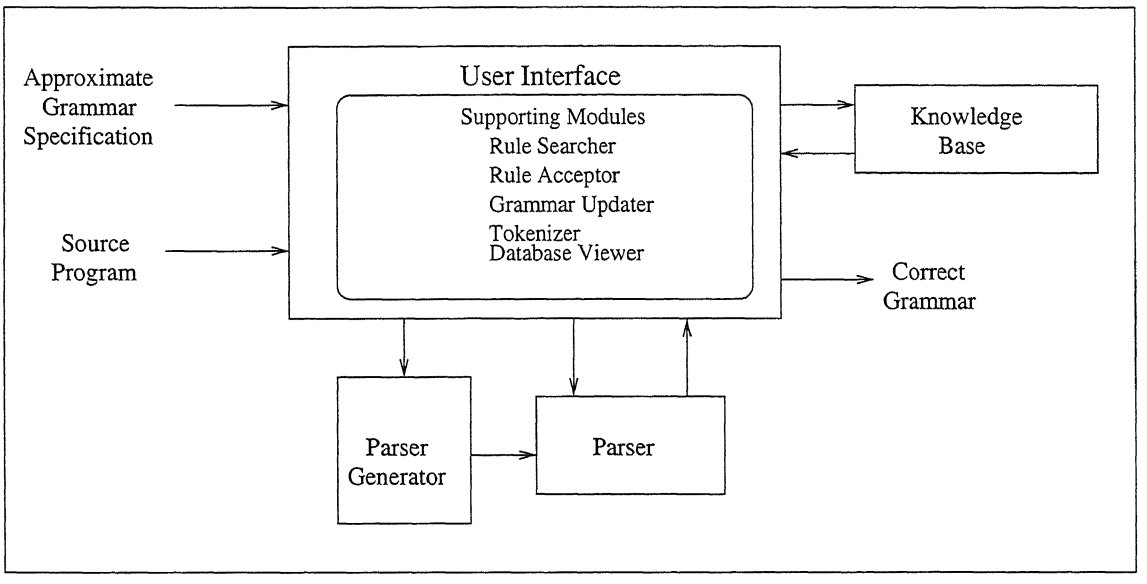


Figure 3.1: Architecture of the Tool

The Parser: The parser is used to parse the input source program. It finds the new constructs in the program for which grammar is not available. We use an automatic parser generator (*Yacc*) to generate the parser.

The Knowledge Base: The Knowledge Base is a repository of grammar rules. It is searched to find a production rule for the new construct in the source program. The new production rules entered by the user are also inserted in the Knowledge Base, so it grows with time.

The GUI Driver: The driver is the component that glues the other two components of the system. It provides a graphical user interface for user input and output. It also includes some modules for grammar updating, quality checking, browsing the Knowledge Base etc.

The Rule Searcher: This is component is invoked when the parser encounters an error. The users can look at the rules suggested to him by the search routine

or he can search the Knowledge Base on the basis of keywords or category of rules.

In the following sections, we describe each of these components in detail.

3.2 The Parser

The parser is one of the core components in the system. It is the component that finds the deficiencies in the grammar. The parser takes a source program and tries to parse the program. If the program is parsed successfully, it means that the input grammar is complete for the given source program. In the case of an error while parsing, the parser returns the error string so that the system can search for a missing rule.

3.2.1 Implementation

We use a LALR(1) parser to parse the source programs. We generate the parser automatically from the grammar specifications using the parser generator tools *Lex*[10] and *Yacc*[7]. Both both the input approximate grammar to the system and the output correct grammar are in the form of *Lex/Yacc* specifications. The *Lex* specifications contain the lexical definitions for the tokens of the language. The *Yacc* specifications contain the production rules for the language grammar together with some C code used for various compilation tasks.

It is easy to get the language grammar from its *Yacc* specifications as the specifications are written in a BNF like notation.

3.3 The Knowledge Base

The Knowledge Base is the most important resource in our approach and an essential component of the system. The Knowledge Base stores grammar rules in ageneric format. The same Knowledge Base can be used for many languages like C, C++, Java, Algol, COBOL, etc.

3.3.1 Structure

The rules in the Knowledge Base are stored in *Backus-Naur Form* (BNF) notation. Each record in the Knowledge Base contains the following four fields:

Search Keyword(s): This field contains the keyword(s) that can be used to search the rule. For example, an *if* statement can have the keywords *if* and *else*.

Construct Category: This field contains the category of the construct. The category can also be used for searching the rule. For example, a *while* statement and a *for* statement both will have the category *iteration*.

The Rule: This field contains the rule itself for the construct in BNF notation. The nonterminals are enclosed in angular brackets.

3.3.2 Implementation

There are many database engines available in public domain, which can be used to implement the Knowledge Base, like *postgres* available on *Linux*. The current approach of using a flat file for Knowledge Base should suffice for most situations as we expect that the approximate grammar to be almost complete and the Knowledge Base will need to contain only a few rules. In our experience the Knowledge Base has of the order of about 20 rules. If, however, Knowledge Base increases to thousands, then it would be better to maintain it as a database. For that we have to suitably modify the interfacing of the system with the Knowledge Base.

3.3.3 Searching

Our algorithm searches the Knowledge Base and returns the closest matching rule. The user is free to select or reject this rule. He can provide his own query for searching the Knowledge Base. The query can be given in two ways: by giving the keyword for searching or the category that the construct falls in. The Knowledge Base returns a list of matching rules. The Knowledge Base provides two more kinds of queries. The first query is issued to retrieve all the category types stored in the Knowledge Base, it is used to help the user while entering a new grammar rule for a construct. The other query is used to retrieve all the grammar rules in the Knowledge Base, it is

used for browsing the knowledge base. This feature is used by the Database Browser provided by the tool. Figure 3.2 shows a snapshot of the Database Browser.

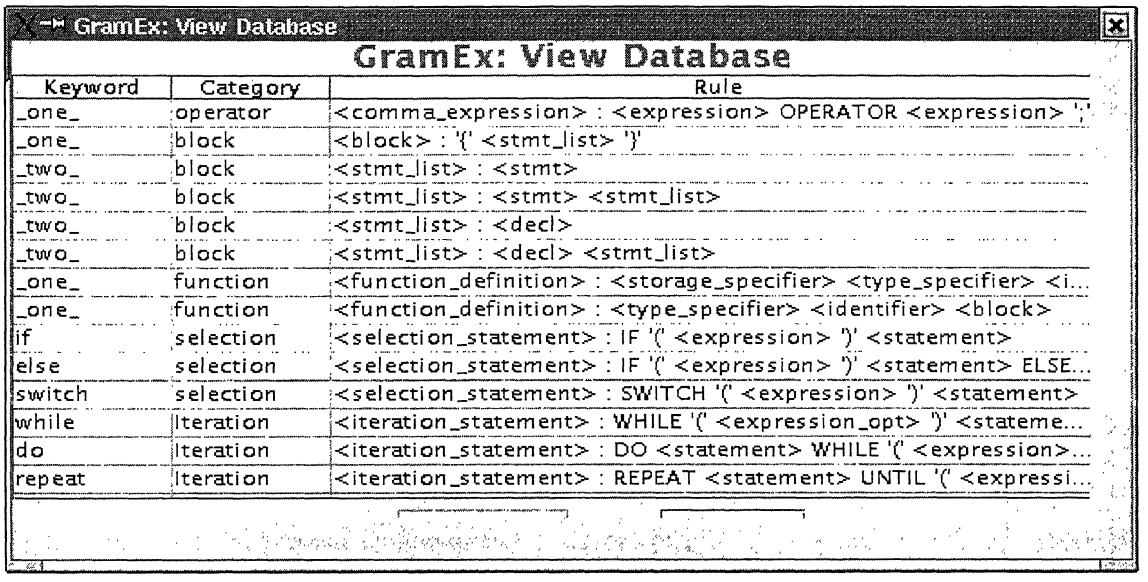


Figure 3.2: A Snapshot of the Database Browser

The insertion of a rule in the Knowledge Base is very simple, the rule is just appended in the file containing rules for the language class. For efficient searching of rules, the Knowledge Base should be indexed both on keyword and category.

3.4 The Driver (User Interface)

The driver integrates the other components of the system. It also provides a graphical user interface. All input from the user, like the grammar specification files, source programs, new rules, lexical definition for new tokens etc. are accepted by the driver. It also keeps the user updated by displaying messages about the progress in processing. Figure 3.3 shows a snapshot of the tool in action.

Apart from providing the a graphical interface to the user, driver is also responsible for various other tasks. Below we discuss some of them, the complete working of the system is described in 3.5.

Tokenizing Yacc specifications: As discussed above, the driver takes the grammar specifications for the language as input. The tokenizer unit of the driver

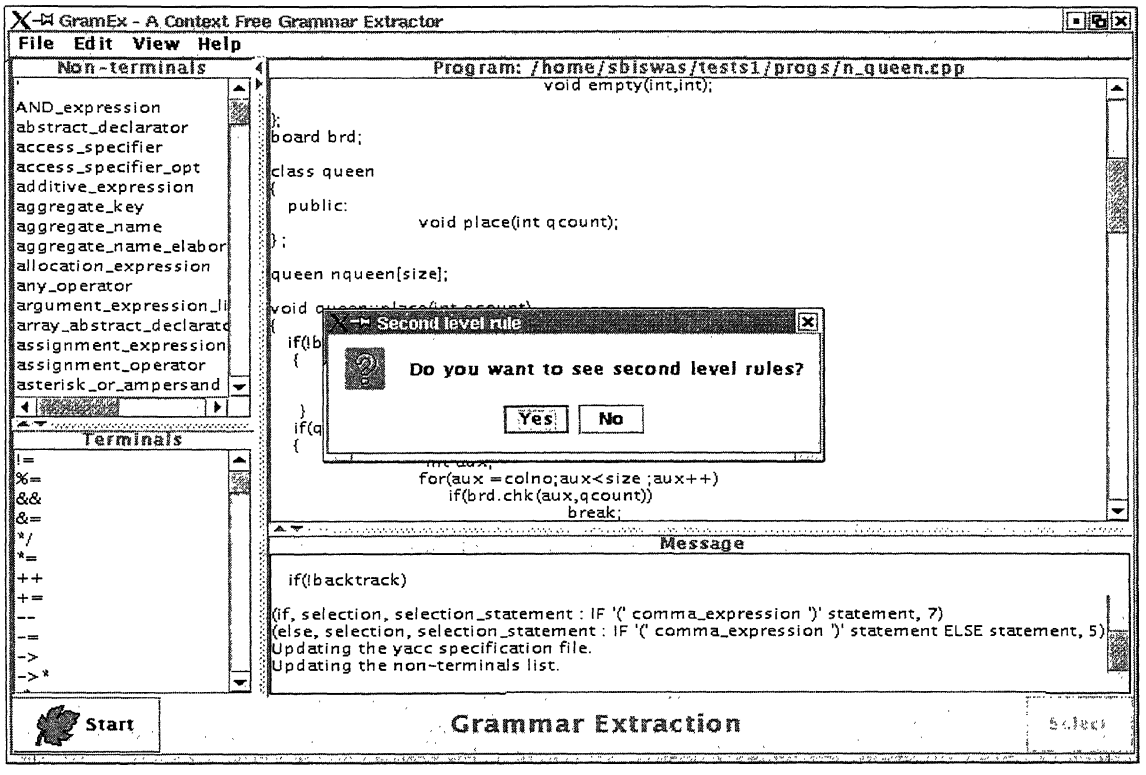


Figure 3.3: A Snapshot of *GramEx* in Action

scans the *Yacc* specification file and extracts all the grammar symbols. These symbols are divided into two groups namely, terminals and non-terminals. A list of these symbols is displayed in the GUI as shown in Figure 3.3. These symbols are helpful to the user while writing new rules or modifying rules before adding them in the grammar.

Getting rules from the user: When no matching rule is found in the Knowledge Base for a new construct, the user is asked to write a rule. Some help from the tool is provided to the user for writing the rule. Hint towards the definition of new tokens is also given by the system. The user can enter a comment with each new rule, the comment is also added in the grammar specifications along with the rule. The new rule is inserted in the Knowledge Base as well.

Updating the grammar specification: After obtaining the grammar rule for the new construct and lexical definition for the new tokens, the grammar specification files are updated. The lexical definitions for the new tokens are added in the *Lex* as well as *Yacc* specification files. The new grammar rule is added

in the *Yacc* specification file at an appropriate position. The specifications are searched for an existing rule with the same LHS as the new rule. If such a rule is found it is updated by appending the RHS of the new rule in the RHS of the existing rule. Otherwise, the new rule is added as it is in the specifications.

Checking the grammar: The new rules added in the grammar may introduce new symbols that are undefined. After updating the grammar, the tool checks whether all newly introduced nonterminals have been defined in the `yacc` file also. Otherwise `yacc` will give error while compiling the parser.

The driver also provides features like Undo, editors for the grammar specifications and source programs, database browser etc. The *Undo* feature can be used to reverse the changes made to the grammar specifications. The modifications need to be undone when some incorrect rules are added in the grammar. The grammar and source program editors can be used to edit the grammar specifications and source program from within the tool. Another feature, the database browser has already been explained in the previous section.

3.5 Working of the Tool

Figure 3.4 shows a flowchart depicting the work flow in the tool. Below, we describe the working of the tool in detail.

The driver accepts the grammar specification files (both *Lex* and *Yacc* specifications) and the source program as input. The tokenizer unit of the driver tokenizes the *Yacc* specifications and extracts the terminal and non-terminal names for helping the user in writing a new rule. Now, the grammar specifications are fed to the parser generator and the parser is generated. This parser is used to parse the source program.

As the grammar is incomplete, the parser will generate an error while parsing the source program. At this point, the user is asked to indicate the correct location of the error. After the line containing the error is found, a keyword based search is made on the Knowledge Base. All the tokens occurring in the line are treated as keywords. The knowledge base returns the matching rules for the new construct. The user is asked to choose the most appropriate rule from this list. If no matching rule is found in the Knowledge Base, the user is asked to write a rule for the new construct. At

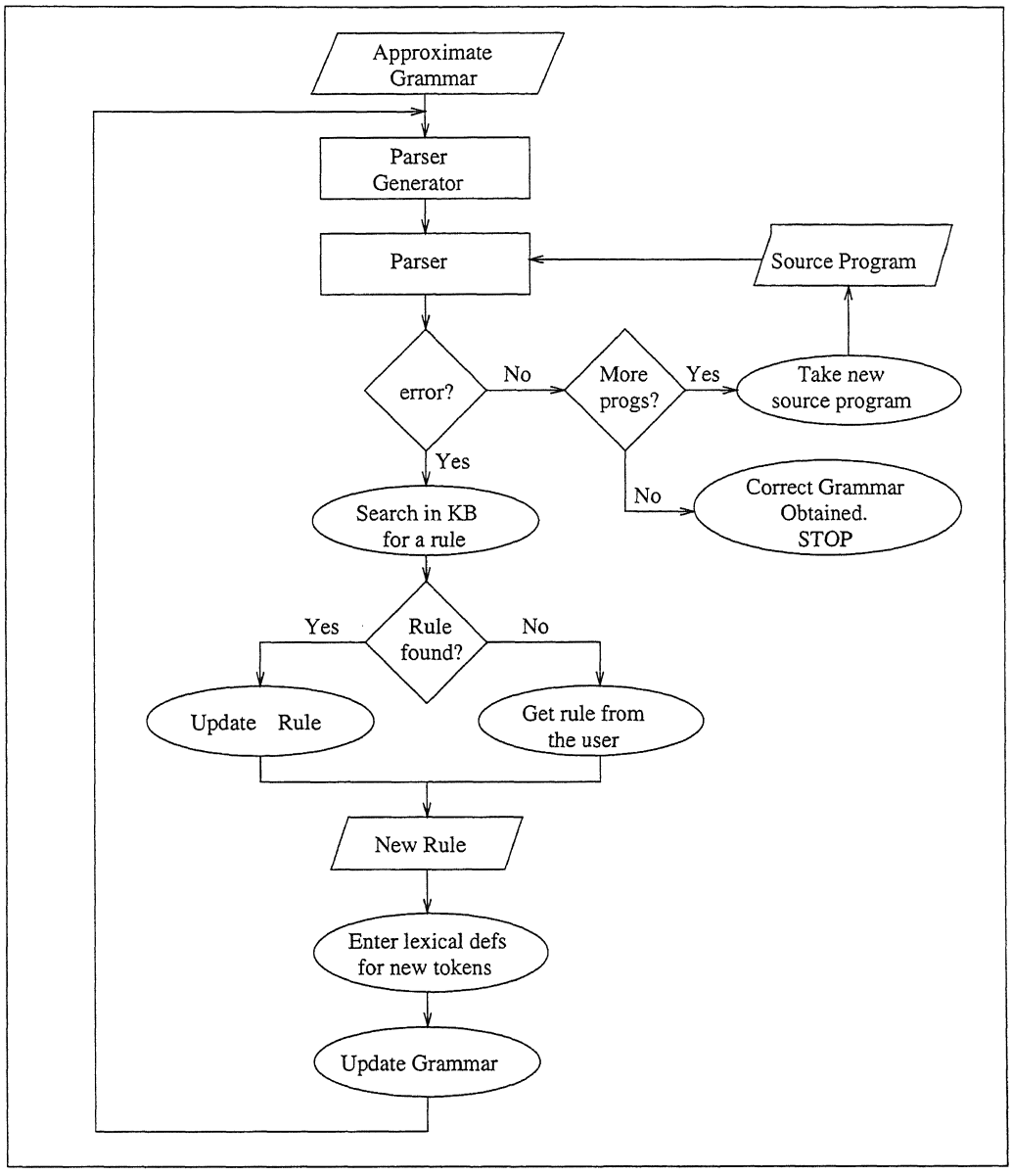


Figure 3.4: Work flow in the Tool

this point, the user can also search the Knowledge Base on the construct category. The definitions for any new terminals and non-terminals introduced in the rule must also be input at this point, either by the Knowledge Base or the user.

After a rule for the new construct is obtained, it should be added in the grammar. The grammar updater unit of the driver updates both the grammar specification files by adding the new rule in the *Yacc* specification file and the lexical definitions for the

new tokens in the *Lex* specification file. In the case when the new rule is written by the user, it is also added in the Knowledge Base.

Now the parser is regenerated from the modified grammar specification files. The program is parsed again from the first line. The whole process described above is repeated until the program is successfully parsed and we get the correct and complete grammar. Another program can be accepted at this point and parsed from the modified parser, in this way the grammar for a set of programs can be obtained.

3.6 Summary

In this chapter, we presented the implementation of the approach for extracting grammar from programs, described in the previous chapter. Our tool called *GramEx* implements the approach.

Chapter 4

Experience, Results and Conclusion

This chapter contains our experience of working with the tool *GramEx* and experimental results obtained by using the algorithms mentioned in chapter2.

4.1 Experience

We have used our approach for programs written in C and C++. We took the complete grammar from [15]. Then we deleted a few rules from it and tried to parse programs with those constructs. At the start of our experiments the Knowledge Base contains the rules shown in Table. 2.1. This experiment was mainly designed to test how efficient our heuristics were in searching for the closest matching rule in the Knowledge Base. Let us consider first a sample program and look at how the Error_String is computed from the modified parser stack. We gave the program shown in Fig. 4.1 as input.

We had removed all the rules related to iteration. The following rules were removed from the grammar.

1. `iteration_statement: WHILE '(' comma_expression_opt ')' statement`

2. `iteration_statement: DO statement WHILE '(' comma_expression ')' ';'`

```
[1]int get_next_tok(char e[MAXOP])
[2]{
[3]     int c,i=0;
[4]     while ((c=postr[ind++]) == ' ')
[5]             ;
[6]     if (c == '\0') return ENDEXPR;
[7]     ind--;
[8]     if (!isdigit(c) && c != '.' && !isspace(c)) {
[9]             while ( c != '\0' && !isspace(c))
[10]                    e[i++] = c = postr[ind++];
[11]            e[i-1] = '\0'; ind--;
[12]            if (i>2) return UNARYOP;
[13]            else return BINARYOP;
[14]    }
[15]    if (isdigit(c))
[16]            for(;isdigit(e[i++] = c = postr[ind++]);)
[17]                    ;
[18]    if (c == '.')
[19]            while(isdigit(e[i++] = c = postr[ind++]))
[20]                    ;
[21]    e[i-1] ='\0';
[22]    ind--;
[23]    return OPERAND;
[24]}
```

Figure 4.1: Sample C Program

3. `iteration_statement: FOR '(' comma_expression_opt ';' comma_expression_opt ';' comma_expression_opt ')' statement`

4. `iteration_statement : FOR '(' declaration comma_expression_opt ';' comma_expression_opt ')' statement`

When the program in Fig. 4.1 was parsed with the incomplete grammar the parser encountered error and the resulting auxiliary parser stack is shown in Fig. 4.2.

When the first while loop (line 4) is executed by the parser as no rule exists in the grammar, our modified parsing technique and heuristics for selecting rules from

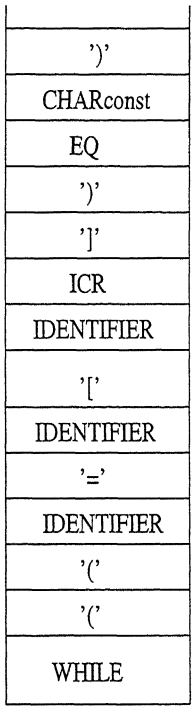


Figure 4.2: Parser stack at the time of error

the Knowledge Base throws up two rules - one for *while* and one for *do-while* (rules 4 and 5) in Table. 2.1. The error string computed by algorithm in Fig. 2.3 is

```
WHILE '(' '(' IDENTIFIER '=' IDENTIFIER '[' IDENTIFIER ICR ']' ')' EQ
CHARCONST ')'
```

So the Error Keyword set is ES = { WHILE }. The *Compute_Proximity* algorithm of Fig.2.5 and 2.6 works as follows. It picks rules one by one and assigns a proximity value to it. Only those rules that have some keywords common with the error string set (ES) will be considered. So only rules 4 and 5 are considered. For rule 4 and 5 the algorithm works as follows.
For rule4 RS = {WHILE}
$\alpha = 1; \delta = 1$; /*the common keyword occurs in the first position in the two sets.*/
P = 4; in line 21
d = 0; in line 33
P = 4 - 0 - 0 = 4;in line 35

For rule5 RS = {DO,WHILE}
$\alpha = 2$; /* In RS WHILE occurs in second position */
$\delta = 1$;
P = 4;
d = 0;
P = 4 - 1 - 0 = 3;

Both the rules are shown. Rule 4 is ranked higher. These rules are generic. They cannot be inserted directly into the grammar file because they contain a different set of nonterminals than that used in this grammar. We had mentioned previously that the user also needs to give a mapping of nonterminals used in the grammar file to that used in the Knowledge Base. The nonterminals we need to replace are `iteration_statement`, `expression_opt`, `expression` and `statement` in the two rules. Since we have the mapping to actual nonterminals used we replace them with `iteration_statement`, `comma_expression_opt`, `expression` and `statement` respectively. So after the replacement the rules look like.

1. `iteration_statement : WHILE '(' comma_expression_opt ')' statement`

2. `iteration_statement : DO statement WHILE '(' comma_expression ')' ';'`

The user selects the first rule and includes it into the grammar. The second level rule suggestions ae as follows:

```
comma_expression_opt :  '(' IDENTIFIER '=' IDENTIFIER '[' IDENTIFIER
ICR ']' ')' EQ CHARCONST
```

The parser is reconstructed and parsing resumes from the first line. This time the `while` loop in line 4 as well as line 9 get parsed successfully. The next error occurs in line 16. The error at the *for* loop will result in two matching rules to be returned from the Knowledge Base. They are rule number 6 and 7. Only one keyword occurs in the error set, `FOR`. Since both rules have the same keyword set RS so both will get the same rank. The user will have to break the tie and chose any one. Thus our heuristic does not work if two rules have same keyword set in the same order. If the order is different then the two rules receive different values of proximity. The user selects rule 6 and the rule is included in the grammar after proper modifications to its nonterminals. The time required for selecting the rule depends on user. The parser needs to be reconstructed from the modified grammar after a new rule is inserted.

4.2 Experiments

The experiments were conducted with C and C++ grammars. The experiments were initially done to find the most appropriate value of the parameter β in the algorithm *ComputeProximity*. The choice of β depends on how many keywords can we expect to find in a rule on average. This would give an indication of the number of keywords present per rule in the Knowledge Base and hence how many keywords may be present in the new construct. We have analyzed two freely available C++ grammars to find number of rules per rule. We took two grammars : C++ grammar by J.Roskind [15] and ISO-C++ grammar. The results are

File Name	roskind.y	ISO.y
Number of rules	665	494
Number of rules with 1 keyword	53	88
Number of rules with 2 keywords	2	5
Number of rules with more than 2 keywords	0	0

Table 4.1: Comparison of two C++ grammars

Since ISO grammar has less number of rules it has more keywords per rule as the number of keywords is constant. There are many keywords which occur in more than one rule. From the table it can be inferred that no rule contains more than 2 keywords. This information has been used to decide the value of parameter β as 4 in the algorithm *ComputeProximity*. We chose 4 as a conservative estimate on the number of keywords that can occur in a rule.

We conducted our experiments by taking a set of 30 C++ and 15 C programs. We used Roskind's [15] grammar as our approximate grammar. Then all the rules related to selection and iteration, a total of 7 rules, were removed. The Knowledge Base consisted of the generic rules shown in table 2.1. The programs we took had the constructs corresponding to *while, do-while* and *for* loops. They also had statements corresponding to selection i.e. *if-else* and *switch-break*. Our tool was then used to recover the deleted rules. The search algorithm returned the correctly ranked rules in six out of the seven cases. The rule where the search algorithm could not make the correct ranking was the rule for *for*. There are two *for* loops in C++ (rules 6 and 7 in Table 2.1) and our algorithm returns both the rules with the same proximity. A

rule which had a match in the Knowledge Base took 6-10 minutes to handle. This includes the time to insert the rule in the Knowledge Base and recompile the parser. In order to recover the complete grammar it required 1 hour 20 minutes.

We took another set of 10 C++ programs and then inserted the following new constructs in the program:

1. `iteration_statement : DOWHILE '(' expression_opt ')' statement`
2. `iteration_statement : REPEAT statement UNTIL '(' comma_expression ')' ';'`

These rules do not have corresponding templates in the Knowledge Base. Thus the keyword based search technique cannot return matching rules. However the first time these constructs are encountered the rules are entered in the Knowledge Base also. But for the first time the user has to create a new rule and this requires more time and effort on part of the user. He needs to have thorough knowledge of the structure of the construct although some help can be obtained from the tool as to which nonterminal to use by looking at the rules of similar structure in the Knowledge Base.

4.3 Conclusion

In order to maintain large software systems we need automated tool support. It has been shown that grammar based tools are more efficient and reliable compared to other tools which do not take the programming language into consideration. Thus it is necessary to extract the grammar of the language in which a software is written.

In this thesis we have provided a parser based approach for extracting grammar from programs. Our approach depends on a Knowledge Base of rules which is searched every time the parser fails to successfully parse a construct. The philosophy behind Knowledge Base is that all declarative, block structured programming languages will use a rules with a fixed structure with small variations from language to language. Thus when the Knowledge Base is made large enough it will be able to answer most of the queries. The approach is sound and has advantages like it can be applied in cases

where the only resource available is the source code of programs. Our experiments also helped us to come up with certain heuristics which may be used to search for the most appropriate rule among many candidate rules. An attempt also has been made to recognize two-level rules where more than one rule need to be inserted for successful parsing.

Bibliography

[1] Dario Bianchi. Learning Grammatical Rules from Examples Using a Credit Assignement Algorithm. In *1st Online Workshop on Soft Computing*, 1996.

[2] Boris Burshteyn. USSA-Universal Syntax and Semantics Analyzer. *ACM SIGPLAN Notices*, 2(1):42–60, 1992.

[3] David N. Chin and Alex Quilici. DECODE: A Co-operative Program Understanding Environment. *Software Maintenance: Research and Practice*, 8:3–33, 1996.

[4] E.M.Gold. Complexity of Automaton Identification from given data. *Information and Control*, 37(3):302–320, 1978.

[5] Philip H.Newcomb and Melvin Scott. Requirements for Advanced Year 2000 Maintenance Tools. *IEEE Computer*, 30(3):52–57, 1997.

[6] Rahul Jain. Extracting Grammar from Programs. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, http://www.cse.iitk.ac.in/research/mtech1998/9811115.html, February 2000.

[7] S.C. Johnson. *Yacc - Yet Another Compiler Compiler.* Technical Report 32, Bell Laboratories, 1975. UNIX programmars manual Vol 2.

[8] Capers Jones. *Estimating Software Costs.* Mc-Graw Hill, 1998.

[9] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[10] M.E. Lesk. *Lex - A Lexical Analyzer Generator.* Technical Report 39, Bell Laboratories, 1975. UNIX programmars manual Vol 2.

[11] M.H.Blaha and W.Premerlani. *Object Oriented Modeling and Design for Database Applications.* Prentice Hall, 1998.

[12] C.Dekkers and A. vanZwol M.J. Nederhof, C.H.A. Koster. The grammar workbench: A first step towards lingware engineering. In *Second Twente Workshop on Language Technology - Linguistic Enginering: Tools and Products*, volume 92-99, pages 103–115, 1992.

[13] Rajesh Parekh and Vasant Honavar. Automata Induction ,Grammar Inference and Language Acquisition, 2000. Invited chapter in Handbook of Natural Language Processing.

[14] Charles Rich and Linda M. Willis. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, January 1990.

[15] Jim Roskind. C and c++ Yacc files. Available at: *ftp://iecc.com/pub/file.*

[16] Spencer Rugaber. White Paper on Reverse Engineering. Available at: *http://www.cs.gatech.edu/reverse/repository/white_paper.ps.*

[17] P.S.Paolucci S.Cabasino and G.M. Todesco. Dynamic Parsers and Evolving Grammars. *ACM SIGPLAN Notices*, 27(11):39–48, 1992.

[18] Alex Sellink and Chris Verhoef. Towards Automatic Modification of Legacy Assets. Available at: *http://adam.wins.uva.nl/ x/aml/aml.html.*

[19] Alex Sellink and Chris Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings of the 13th IEEE International Conference on Software Engineering*, 1998.

[20] Alex Sellink and Chris Verhoef. Generation of Software Renovation Factories from Compilers. In *Proceedings of the International Conference on Software Maintenance*, 1999.

[21] Alex Sellink, Chris Verhoef, and Mark van den Brand. Generation of Components for Software Renovation Factories from Context Free Grammars. In *Proceedings of the 4th Working Conference on Reverse Engineering*, 1997. Available at: *http://adam.wins.uva.nl/ x/trans/trans.html.*

[22] W.M Ulrich. The evolutionary growth of software reengineering and the decade ahead. *American Programmer*, 3(11):14–20, 1990.